Bringing Forensic Readiness to Modern Computer Firmware

Tobias Latzo, Florian Hantke, Lukas Kotschi and Felix Freiling*

Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany

ARTICLE INFO

Keywords:
UEFI
memory acquisition
forensic readiness
firmware

ABSTRACT

Today's computer systems come with a pre-installed tiny operating system, which is also known as UEFI. UEFI has slowly displaced the former legacy PC-BIOS while the main task has not changed: It is responsible for booting the actual operating system. However, features like the network stack make it also useful for other applications. This paper introduces UEberForensIcs, a UEFI application that makes it easy to acquire memory from the firmware, similar to the well-known cold boot attacks. There is even UEFI code called by the operating system during runtime, and we demonstrate how to utilize this for forensic purposes.

1. Introduction

The role of main memory analysis has become a more and more important part of digital forensic investigations. Main memory often contains valuable data that is never written to persistent storage. Examples are running processes, open network connections or even encryption keys that are necessary to decrypt containers. Furthermore, there is fileless malware that cannot be detected otherwise.

Memory acquisition has, therefore, become an essential part of forensic investigations. Ideally, memory acquisition cannot be detected by the target system, does not change data, and is performed atomically [21]. To provide the highest level of authenticity, memory acquisition should be performed on as low a level as possible, i.e., below the operating system (OS) [8]. However, main memory acquisition is much more intricate than acquiring hard disk contents. Nowadays, there are numerous ways of memory acquisition [20], e.g., using kernel support or modules like the Linux tools Pmem [16] or LiME [17]. Virtualized target systems can often be halted and the memory acquired using built-in tools of the hypervisor. It is even possible to virtualize the target system on-the-fly [10]. However, most of these techniques have the following requirements: (1) Forensic software has to be deployed on the target system at runtime, and (2) this software has to run with root privileges. Moreover, even if these requirements can be met, there are anti-forensic techniques that may circumvent or tamper with memory acquisition [13, 15]. In general, it would therefore be useful if forensic software was "pre-installed" on the system, a condition known as *forensic readiness* [14].

The *Unified Extensible Firmware* (UEFI) was introduced as the successor of the meanwhile nearly 40 years old PC-BIOS and is a "pre-installed" software, albeit not for forensic purposes. UEFI allows to start the OS in long mode, supports Secure Boot, and even own EFI applications can be executed in the UEFI Shell. Most UEFI implementations even come with a full network stack and sometimes even with a web browser. UEFI also specifies Runtime Services (RTSs) that can be called by the OS. These allow, for example, reading and setting UEFI variables or updating the firmware. There is a UEFI reference implementation called *EFI Development Kit II* (EDK II) [18].

In this paper, we exploit modern computer firmware's high capabilities and bring forensic readiness to the UEFI. For this, we introduce

UEFI built-in memory forensics

(abbreviated as UEberForensIcs) for which we integrated forensic memory acquisition software. That can be used during the boot process. The memory acquisition is based on the concept of *cold boot*, which is explained in more detail below. Furthermore, we show how to persist code in the UEFI RTSs and get code execution that can also be used for forensic software. Additionally, we have built a tracer that traces calls of UEFI RTSs.

1.1. Related Work

In 2008, Halderman et al. [6] introduced *cold boot* attacks. They exploited the fact that DRAM modules are not instantly cleared when unplugged, which is also known as the memory remanence effect. To acquire memory, they transplanted DRAM modules and attached it to an analysis system where the module is readout. During the replug procedure, the RAM module is cooled with coolant spray. Eventually, the researchers were able to restore encryption keys.

While the focus of Halderman et al. [6] was key recovery after memory transplantation, the authors also mentioned the possibility of hard resets and booting a system for memory acquisition. In the latter, one has to deal with the BIOS footprint that overwrites few megabytes of the RAM. Further research [5] revealed that newer RAM modules are not as vulnerable to memory transplant attacks as the older DDR2 modules. Furthermore, newer modules do scramble data to avoid the parasitic effects of semiconductors. However, memory scrambling has not proven to be effective protection against cold boot attacks [1].

Regarding related work on UEFI, we are aware of a Master's thesis [9]. The students made use of a signed UEFI

*Corresponding author

tobias.latzo@fau.de (T. Latzo); florian.hantke@fau.de (F. Hantke); lukas.kotschi@fau.de (L. Kotschi); felix.freiling@cs.fau.de (F. Freiling)

ORCID(s):

application that was used to dump physical memory to a USB flash drive. They focused on building a static chain of trust whereby the trust anchor is the firmware. Furthermore, there is a blog post by Frisk who used the UEFI RTSs to circumvent 4 GiB DMA limitations in PCILeech [3]. Usually, the Linux kernel is mapped into the upper physical memory. So PCILeech cannot inject code via 32 bit DMA. To get around this limitation, he manipulated the UEFI RTSs function pointer table — that is located in lower memory regions — to inject own code.

ISO/IEC 27043:2015 [7] defines *digital forensic readiness* as the "process of being prepared for a digital investigation before an incident has occurred." Forensic readiness is related to preparation phases in many process models of incident response and digital forensic investigations and usually involves establishing a capability for securely gathering legally admissible evidence in case of an incident [14]. In practice, the quality of forensic readiness is closely related to the level of logging, the effectiveness of alerting and incident management processes and the quick availability of evidence acquisition capabilities which ideally are predeployed as software [11] or in hardware [2].

1.2. Contribution

This paper shows how to make a computer's UEFI forensic ready. The main contributions of this are as follows:

1. We introduce UEberForensIcs show how to integrate forensic software that enables cold boot like memory acquisition directly into a computer's firmware. The evaluation in this paper reveals that this approach can also be practically used.
2. Furthermore, we show how to persist code in the UEFI that is executed when the operating system is running. This code runs with kernel privileges and can also be used for memory acquisition.
3. We developed an OS-independent RTS tracer. The RTS are thereby traced in the RTS itself. Our evaluation gives insights which and how often specific RTS are typically called in different scenarios.

We have published UEberForensIcs [1] and the RTS tracer [2] on Github.

1.3. Outline

In Section 2 background information about the UEFI and EDK II is given. The architecture and setup of our experiments is described in Section 3. Then, we show insights into the implementation of the built-in forensic acquisition software including an evaluation. In Section 5 we show how the RTS tracer runs with hooking. Finally, in Section 6 this paper is concluded.

2. Background

In the following, we want to give some background information on some existing concepts that are used in this paper.

2.1. Criteria for Memory Acquisition

Vömel and Freiling [21] defined three criteria for forensically sound memory acquisition: *correctness*, *atomicity* and *integrity*. In the following, we want to briefly explain these criteria and show how to quantify them [4].

A memory snapshot is considered to be *correct* if the used memory acquisition software acquired the memory's actual content. Obviously, this criteria is very fundamental for memory acquisition.

Memory acquisition software that is running on the target itself usually cannot stop all other system activity. This may lead to memory dumps that show the effects of events for which the actual cause has not been recorded. If such inconsistencies do not occur — this is usually the case if the system can be halted — a memory dump is called *atomic*. Gruhn and Freiling [4] quantified the atomicity of a memory dump by the time between the acquisition of the first memory region and the last memory region.

Integrity is ensured if the content of a memory image is not changed after an investigator decides to take a snapshot. Vömel and Freiling [21] state that integrity can be quantified by the level at which the process of taking the snapshot changes memory.

2.2. Unified Extensible Firmware Interface

The UEFI was introduced in 1998 as a successor of the legacy PC-BIOS. Often the UEFI is still called BIOS. A more generic term for UEFI and BIOS is *firmware*. The UEFI boots itself into protected mode (32 bit) or long mode (64 bit) instead of the real mode (16 bit). Obviously, this makes development much easier. As a consequence, the UEFI implementations are often tiny OSs with own applications, network stack, and so on. There are several specified stages, e.g., the Security (SEC) phase as the first stage, followed by the Pre EFI initialization (PEI) and *Driver Execution Environment* (DXE) phase. When reaching the DXE phase, basically all hardware initializations happened, and hardware can be used.

In 2004, Intel released an open-source implementation called Tiano of an EFI. Tiano evolved to EDK II and is now maintained by the TianoCore community [18].

3. Architecture and Setup

Developing and debugging firmware is an intricate affair and usually requires a special setup to be performed. We now describe the setup in which we developed our system and performed the experiments.

Figure 1 shows a simplified graph of the architecture we use for our experiments. Filled boxes indicate that these modules are our own developments. The target is running in a virtual machine with QEMU hypervisor. This makes development easier because, in this case, we do not need to reprogram SPI flash chips for every change. Furthermore, it simplifies debugging. The right side of the graph is dedicated to the built-in cold boot part (see also Section 4) while the left side is dedicated to the runtime forensics part (see also Section 5).

[1] https://github.com/ueFAUrensics/UEberForensIcs
[2] https://github.com/ueFAUrensics/RTStracer

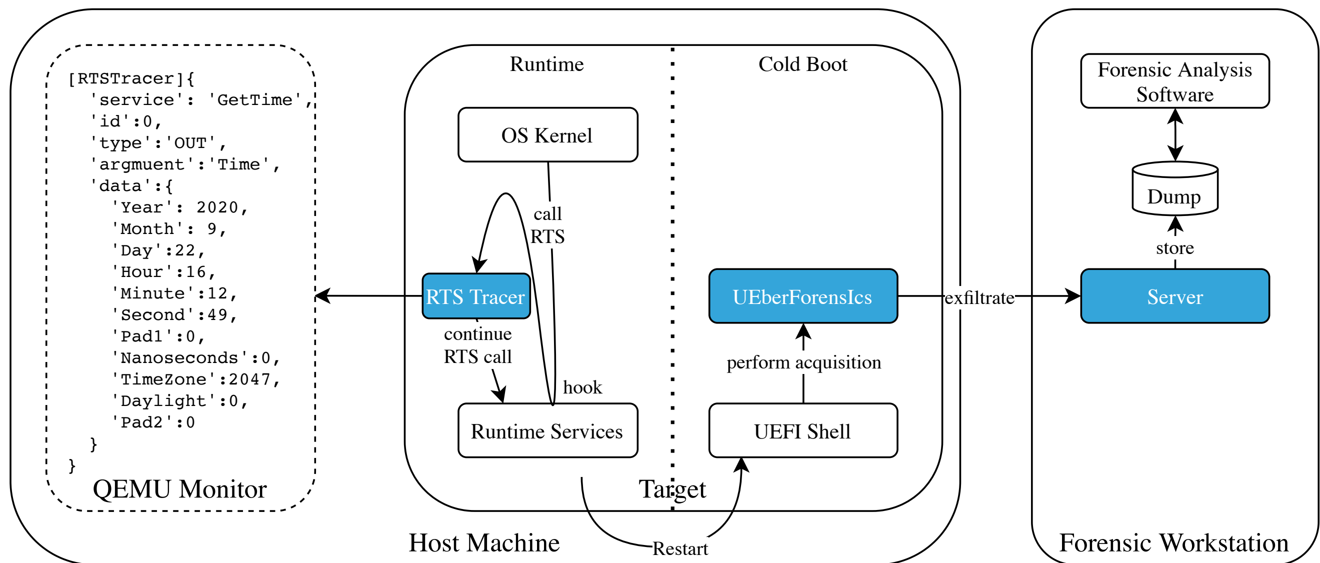


Figure 1: Simplified architecture of UEberForensIcs and the RTS tracer.

Table 1
Memory map of the virtual machine we used for our experiments.

#	Start	End	Pages	Size	Purpose
1	0x00000000	0x0009ffff	160	640 kiB	System RAM
2	0x000a0000	0x000bffff	32	128 kiB	PCI Bus
3	0x00100000	0x7fffffff	524032	2047 MiB	System RAM

3.1. Hardware Setup

The entire research was conducted on a standard laptop with an Intel Core i5-5200U CPU (2.20GHz, 2 cores) and 8 GiB of RAM. It runs Ubuntu Linux 18.04.4 with kernel version 4.15.0-118. The installed QEMU version is 4.2.92.

3.2. VM Setup

Usually, virtual machine monitors come with specialized firmware implementations. In most cases, emulating firmware is not intended. For virtual machines, there is a target for EDK II called OVMF. This port supports QEMU's virtual hardware. In Listing 1 one can see the command to start the corresponding QEMU *Virtual Machine* (VM). The VM is running Ubuntu Linux 20.04 and has 2 GiB of RAM.

```
qemu -bios edk2/Build/OvmfX64/RELEASE_GCC5/FV/OVMF.fd
    -drive format=raw, file=ubuntu-linux.raw
    -drive format=raw, file=fat:rw:vm-content
    -global virtio-net-pci.romfile=""
    -nic tap,model=virtio-net-pci
    -m 2048M
    -debugcon file:debug.log
    -enable-kvm -cpu host
    -cdrom ubuntu-20.04-desktop-amd64.iso
```

Listing 1: Start of the virtual machine using QEMU.

Table 1 shows the physical memory map of our virtual machine. In this case, memory is quite cohesive. Memory range #3 is by far the largest memory region. A real system's RAM is usually more fragmented than those in QEMU. There is only a single memory hole from #2 to #3.

4. Built-in Cold Boot

In this section, we give insights into the implementation of UEberForensIcs. UEberForensIcs is a forensic cold boot like acquisition software that is integrated into the firmware.

The use case of UEberForensIcs is that it is pre-installed on the firmware of a computer. While the OS is running, a potential incident happens, and so an incident responder is alerted. The incident responder wants to analyze what happened on the system with memory analysis. For the acquisition, they restart the computer into the EFI Shell and perform memory acquisition using UEberForensIcs. Therefore, the analyst needs no special equipment or installed tools on the host. The dump is transferred via network to the Forensic Workstation where it can be analyzed.

4.1. Implementation

UEberForensIcs can be used as a standalone application or a dynamic command. For the evaluation, we used the latter variant. Basically, UEberForensIcs is implemented as a DXE driver.

We do not save the memory dump on the local drive because that would lead to corruption. Instead, we exfiltrate the data via the network (see also Figure 1). So UEberForensIcs requires an active network connection to the Forensic Workstation, and so we make use of EDK II's TCP stack. The IP address is obtained via DHCP. When the connection is established, UEberForensIcs traverses the memory and sends it page-wise to the Forensic Workstation.

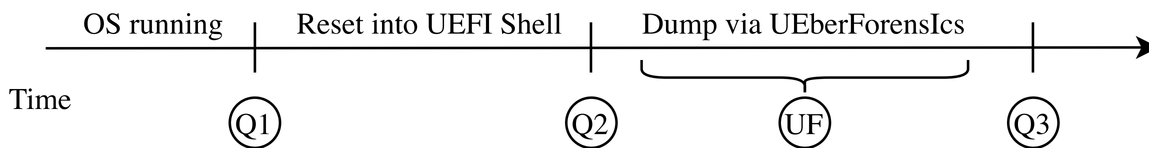


Figure 2: Timeline of the system with the four memory dumps.

4.2. Evaluation

Gruhn and Freiling [4] provided a framework to evaluate memory acquisition tools in terms of *correctness*, *atomicity* and *integrity* [21]. Basically, cold boot like attacks are performed atomically, i.e., the RAM module is removed, or there is a hard reset. So in this evaluation, we want to focus on correctness and integrity. For this purpose, we have generated the following four memory dumps, which one can see in Figure 2. The dumps are acquired in the following way:

Q1 The first dump is acquired using QEMU's `pmemsave` feature while the OS is running. Before `pmemsave` is started, the system is paused. We consider this dump as the *ground truth*.

Q2 The second dump is also performed using QEMU's `pmemsave` after the reset when the EFI Shell is started. To acquire memory atomically, the system is also paused. Furthermore, the OS is not running anymore. This means that processes that have run before do not alter memory anymore. However, EDK II also overwrites some smaller parts of memory.

UF The third dump is generated with our tool. Since the dump is performed on the same system, we cannot pause the system. However, we consider it to be atomic because the processes and any other thread of the OS processes are not running anymore. The only running activities belong to EDK II and so are not important.

Q3 The last dump is acquired using QEMU's `pmemsave` after the UEberForensIcs dump is completed.

The evaluation of correctness and integrity is based on the analysis of differing bytes and pages of different dumps. In Table 2 one can see the results of pairwise dumps. A visualization of page-wise (4 kiB) diffs can be found in Figure 3. A blue pixel indicates that the corresponding page is the same in both dumps. A red pixel indicates that the corresponding pages are differing by at least one byte. There are 1024 rows with 512 pages per line, i.e., 2 MiB per line. Addresses are growing from left to right and from the bottom to the top.

In the following two sections, we use these results to show to what extent UEberForensIcs affects memory and argue that UEberForensIcs works properly.

4.2.1. Correctness

First, we want to show that UEberForensIcs works properly. To show this, we compare the ground truth (Q1) with the UEberForensIcs dump (UF), i.e., $\mathit{diff}(Q1, UF)$ and with the dump Q3, i.e., $\mathit{diff}(Q1, Q3)$. It is striking that the total differing numbers are in the same order of magnitude (see also Table 2). Furthermore, the corresponding diffs' visualizations in Figure 3b and Figure 3c show that the diffs are very similar. The ranges of diffing memory regions are basically the same for all dumps. Note that the dump of UF is made sequentially and transferred via network. The dump of Q3 is acquired atomically after the execution of UEberForensIcs. So, these diffs are not completely the same.

The $\mathit{diff}(Q2, UF)$ shows that the QEMU `pmemsave` dump and the UEberForensIcs dump are differing in about 5 MiB. The corresponding visualization in Figure 3d also reveals that the corresponding memory regions are basically the same. For $\mathit{diff}(UF, Q3)$ the diff is around 5.8 MiB. However, as Figure 3f shows, the differing pages are in the upper memory regions as before.

The comparisons of the diffs showed that the dump of UEberForensIcs looks reasonable. Basically, the only differing pages are located in the upper memory regions that we can also observe with the QEMU built-in `pmemsave`.

4.2.2. Integrity

Memory acquisition using UEberForensIcs is performed on the target system. This means that we do change memory. In this section, we show how much memory is changed. We also show which parts of the memory get changed by UEberForensIcs.

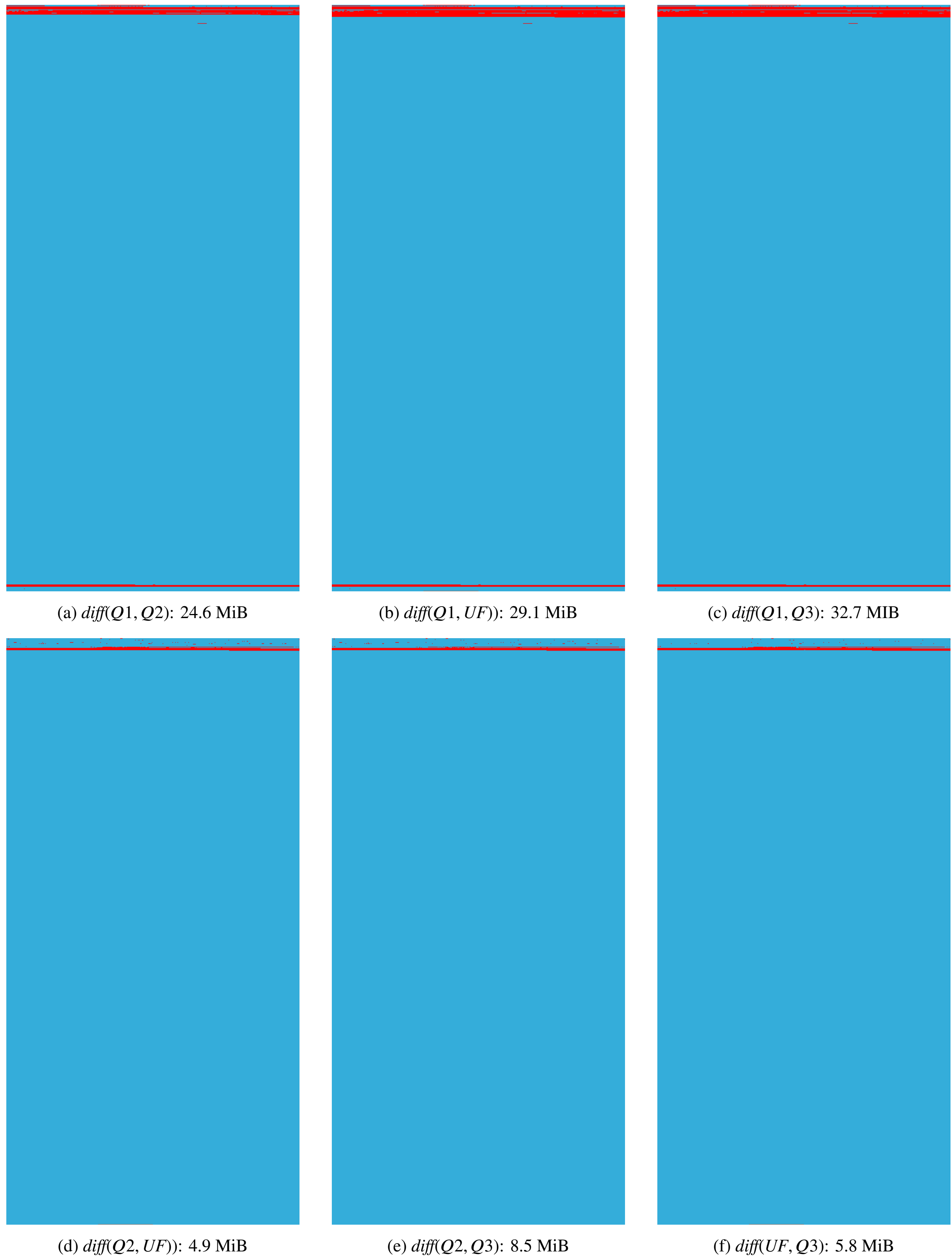
Figure 3 gives a good impression of what and how much memory is changed when using UEberForensIcs. All diffs with the dump when the OS was running (Q1) show that some memory is overwritten in the lower memory regions — basically starting at 0x1000000 — when the computer is reset. This region has a size of about 7 MiB. When the system is restarted, there is no change in this memory region anymore (see also Figure 3d, Figure 3e and Figure 3f).

Furthermore, Figure 3 also shows that the reboot of the system has the most impact. The execution of UEberForensIcs also has impact (see also Figure 3d, Figure 3e and Figure 3f). However, most of these memory regions are changed because of the reboot anyway.

Overall, we can say that the execution of UEberForensIcs changes about 32 MiB of the whole memory. Thereof the most considerable amount is overwritten because of the reboot that loads the firmware. The majority of the firmware in our environment was located in the upper memory regions. There was no single differing byte in the middle of the memory.

4.3. Discussion

Writing software for UEFI is much easier than for the former PC-BIOS. EDK II code is written in C, and there are



(a) $diff(Q1, Q2)$: 24.6 MiB (b) $diff(Q1, UF)$: 29.1 MiB (c) $diff(Q1, Q3)$: 32.7 MIB

(d) $diff(Q2, UF)$: 4.9 MiB (e) $diff(Q2, Q3)$: 8.5 MiB (f) $diff(UF, Q3)$: 5.8 MiB

Figure 3: Visualization of the page-wise diff. Addresses are growing from the left to the right and from the bottom to the top.

Table 2
The table shows the results of differing bytes of the dumps and the corresponding proportion of total memory that is changed.

#	Dump 1	Dump 2	Total Pages	Total Size	Proportion
1	Q1	Q2	8143	24.6 MiB	1.2 %
2	Q1	UF	10245	29.7 MiB	1.4 %
3	Q1	Q3	10260	32.7 MiB	1.6 %
4	Q2	UF	2634	4.9 MiB	0.2 %
5	Q2	Q3	2568	8.5 MiB	0.4 %
6	UF	Q3	2588	5.8 MiB	0.3 %

many features like a full network stack that facilitate the development of their own software.

The evaluation showed that UEberForensIcs is working correctly. However, the whole acquisition process using UEberForensIcs changes about 30 MiB of RAM. This may differ from setup to setup depending on the firmware's footprint. In our setup, the memory ranges that were changed are located on the upper and lower border of the RAM, and so we argue that memory acquisition using UEberForensIcs is practical since we can acquire most memory atomically and do not rely on software on the host that may be manipulated by malware.

However, there are also countermeasures for cold boot attacks. RAM reset on reboot [5], memory scrambling [1] or locking the firmware that an adversary cannot boot from an own device are three examples. In our scenario, we control the firmware. So we can control that such countermeasures are not implemented or are only effective during normal reboots and not when an analyst is present. A particular hardware device could indicate this.

5. Runtime Service Forensics

In the previous section, we have seen how our UEFI driver can be used to perform cold boot attacks. Now, we want to provide the first steps towards forensic memory acquisition using UEFI drivers at runtime. Thus, incident response teams could extract memory without rebooting and installing any memory acquisition software on the target that would change evidence.

In the following sections, we describe how to persist forensic tools in the UEFI RTSs and gain code execution. For this, we provide a proof-of-concept tool that traces all RTS calls. Basically, this technique can also be used to perform memory acquisition in the RTSs.

5.1. Implementation

Same as the former tool, the runtime tool is developed as a DXE driver. However, this time the driver needs to continue execution even after `ExitBootServices()` is called, something common drivers do not fulfill. Therefore, the driver needs to be of the type `DXE_RUNTIME_DRIVER`.

Runtime drivers start in the DXE phase and continue executing after the boot process is finished when the OS is running. Also, they have access to both RTS and boot services. Boot services stop to work after the OS loader calls `ExitBootServices()`, RTSs persist after the DXE phase. Their pointers get converted from physical addresses to virtual ones when `SetVirtualAddressMap()` is called.

To execute code after the DXE phase, the driver needs to be called by another instance, such as the OS. The OS in our research is an instance we do not control. Hence we decided to take RTSs that are called by the OS as a trigger for code execution. To activate the code execution, we implemented hooks for all RTSs and set them in the DXEs phase when it initializes our driver. Therefore, the driver stores the origin service pointer and replaces its address table entry with a pointer to our hook. Furthermore, it registers a notifier to react when the OS calls `SetVirtualAddressMap()` and converts all pointers.

The hooks allow us to execute arbitrary code at runtime, which we developed further to implement an open-source RTS tracer. With the tracer, we can follow the called services and view their arguments to analyze the UEFI behavior thoroughly. The tracer outputs its information in JSON format.

An example call can be seen in Listing 2. Every JSON object contains one argument of the called service and its data. The JSON is limited to a maximum of 255 characters, which is why not all arguments fit in one object. Additionally, every argument is either of the type INput or OUTput and accordantly listed before or after the origin call. The example shows two output arguments of the service *GetTime*.

5.2. Evaluation

The evaluation of the runtime service section is split into two parts. First, we show that we have arbitrary code execution at any time at runtime. Second, we evaluate the data created by the RTS tracer and compare various scenarios.

As mentioned before, we make use of runtime service hooks to execute code in UEFI. UEFI code execution at runtime could be used by incident response teams to extract memory without rebooting the OS. A requirement for this is that it can be executed at any time. However, our trigger depends on RTSs being called, which is not often the case after the user logged in. Nevertheless, our tests show that the OS calls the RTS `GetVariable` whenever the user reads the *efivars* (`/sys/firmware/efi/efivars/`). To prove this, we successfully modified our hook to force `System_Reset` as soon as we read *efivars*. Thus we fulfill the requirement to execute code at any time at runtime, which finishes the first part of

```
[RTSTracer]{
  'service': 'GetTime',
  'id':0,
  'type':'OUT',
  'argmuent':'Time',
  'data':{
    'Year': 2020,
    'Month': 9,
    'Day':22,
    'Hour':16,
    'Minute':12,
    'Second':49,
    'Pad1':0,
    'Nanoseconds':0,
    'TimeZone':2047,
    'Daylight':0,
    'Pad2':0
  }
}
[RTSTracer]{
  'service': 'GetTime',
  'id':0,
  'type':'OUT',
  'argument':'Capabilities',
  'data':{
    'Resolution': 0,
    'Accuracy': 0,
    'SetsToZero':0
  }
}
```

Listing 2: The log shows an example result provided by the RTS tracer.

the evaluation.

For the second part, we evaluate the RTS tracer. Therefore, we recorded the RTS calls on our Ubuntu VM in six different scenarios:

- *Boot* - We started the VM but did not log in.
- *Login* - We started the VM and logged in as the user.
- *Working* - We started the VM, logged in as the user, and performed standard working tasks for 15 minutes. These tasks were reading, writing, and configuring OS settings.
- *Hour* - We started the VM, logged in as the user, and let it run for one hour. The power save mode caused a lock screen, which we unlocked in the end.
- *Switch* - We started the VM and logged in as the user. Afterward, we switched the user.
- *Reboot* - We started the VM, logged in as the user, and rebooted the machine. Then we logged in again.

For every scenario, our RTS tracer collected data. We wrote a parser in Python which is also included in the RTS tracer Github, to analyze and interpret the information. Table 3 shows the summarized results. It shows how often which runtime service was called in each scenario. The table lists only five of the 14 runtime services that are available according to section 5 in the EDK II UEFI Driver Writer's Guide [19]. Even if the guide lists more services, we only observed these five services in all scenarios.

When we look at the different scenarios, we can see that *Login*, *Working*, and *Hour* have the same number of services called. Further, going into more detail, we can see that the calls' arguments are the same every time. This means that the RTSs used during the startup routine remain the same. Moreover, even without studying the EDK II source, we can conclude from the same call number in the three scenarios that during standard OS usage, no RTS is used after the login.

The login and logout processes, on the other hand, make use of RTSs. This is shown by the difference of counted calls in the scenarios *Boot*, *Login*, and *Switch*. We see that 32 additional calls are registered on login and 32 more on logout in the results. All of them are `GetVariable` calls that request either the `OsIndicationsSupported` or the `OsIndications` variable. Both variables tell the OS which UEFI firmware features are supported and activated.

The last scenario, *Reboot*, is different from the previous ones as the counted call number is a lot more. For *GetTime* and *ConvertPointer* the numbers are twice as large compared to the *Login* scenario. This makes sense as we boot the system two times. On the other hand, we count 45 more *GetVariable* calls, 69 more *GetNextVariableName* calls, and 55 less *SetVariable* calls on the second boot process. This is because the second boot process does not register every variable again but uses initialized variables from the first boot process. The variable *OsIndications*, for instance, is only set in the first boot process. Afterward, it is requested 45 times during the first boot and 46 times during the second boot.

As shown above, the RTS tracer works well and gives clear insights into the usage of RTSs.

5.3. Discussion

In this section, we showed how to gain code execution from the UEFI during OS runtime. As a proof-of-concept, we implemented an RTS tracer. The corresponding code is not resident on the hard drive but on the SPI flash chip and copied to the RAM by the system's firmware. However, in contrast to System Management Mode (SMM)-based approaches [12], RTSs are not executed on a higher privilege level but on the same as the OS. Developing code for the RTS is much easier than SMM's 16-bit Real Mode code.

It is also possible to perform memory acquisition from the RTSs. However, the exfiltration of memory is more complicated than in UEberForensIcs. The OS manages the network stack and has configured the network interface card. Other possibilities are to use persistent storage. Neverthe-

Table 3
The table shows the number of RTS calls in various scenarios

Runtime Service	Boot	Login	Working	Hour	Switch	Reboot
GetTime	46	46	46	46	46	92
GetVariable	754	786	786	786	850	1617
SetVariable	110	110	110	110	110	165
GetNextVariableName	499	499	499	499	499	1067
ConvertPointer	91	91	91	91	91	182
Total	1500	1532	1532	1532	1596	3123

less, similar to the network interface, the OS manages hard drives. So it is not easy to use the RTS for memory exfiltration without adapting the OS kernel or drivers.

6. Conclusion and Future Work

In this paper, we introduced UEberForensIcs that brings forensic memory acquisition to modern computer firmware. With UEberForensIcs, an analyst can perform simple cold boot attacks without any craftsmanship. Additionally, the dump can be considered to be atomic. The only precondition is that the UEFI is forensic-ready, i.e., UEberForensIcs must be part of the UEFI before the need for memory acquisition arises.

Our evaluation showed that only small distinct parts of the memory get overwritten by the firmware. For the development and evaluation, we used a QEMU VM. However, future work should also consider compatibility and other possible memory layouts on actual physical systems.

Furthermore, we demonstrated how to gain code execution with kernel privileges without injecting code into the kernel without any persistent file on the hard drive. Therefore, we hook the UEFI RTSs. As a proof-of-concept, we developed an RTS tracer that traces all occurring RTS calls of the OS kernel. The discussion in Section 5.3 yielded that integrating memory acquisition software in the RTSs can be beneficial. However, it is hard to exfiltrate data from there.

Acknowledgments

This research is supported by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Research and Training Group 2475 "Cybercrime and Forensic Computing" (grant number 393541319/GRK2475/1-2019).

CRediT authorship contribution statement

Tobias Latzo: Conceptualization, Investigation, Methodology, Software, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing. **Florian Hantke:** Conceptualization, Investigation, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Lukas Kotschi:** Conceptualization, Investigation, Software, Validation, Writing – review & editing. **Felix Freiling:** Supervision, Methodology, Writing – original draft, Writing – review & editing.

References

[1] Bauer, J., Gruhn, M. and Freiling, F. C. [2016], 'Lest we forget: Cold-boot attacks on scrambled ddr3 memory', *Digital Investigation* **16**, S65–S74.

[2] Carrier, B. D. and Grand, J. [2004], 'A hardware-based memory acquisition procedure for digital investigations', *Digit. Investig.* **1**(1), 50–60.
URL: *https://doi.org/10.1016/j.diin.2003.12.001*

[3] Frisk, U. [2017], 'Attacking UEFI Runtime Services and Linux'.
URL: *http://blog.frizk.net/2017/01/attacking-uefi-and-linux.html*

[4] Gruhn, M. and Freiling, F. C. [2016], 'Evaluating atomicity, and integrity of correct memory acquisition methods', *Digital Investigation* **16**, S1–S10.

[5] Gruhn, M. and Müller, T. [2013], On the practicability of cold boot attacks, *in* '2013 International Conference on Availability, Reliability and Security, ARES 2013, Regensburg, Germany, September 2-6, 2013', IEEE Computer Society, pp. 390–397.

[6] Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J. and Felten, E. W. [2008], Lest we remember: Cold boot attacks on encryption keys, *in* P. C. van Oorschot, ed., 'Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA', USENIX Association, pp. 45–60.
URL: http://www.usenix.org/events/sec08/tech/full_papers/halderman/halderman.pdf

[7] International Organization for Standardization [2015], 'Iso/iec 27043:2015: Information technology – security techniques – incident investigation principles and processes'.

[8] Latzo, T., Palutke, R. and Freiling, F. [2019], 'A universal taxonomy and survey of forensic memory acquisition techniques', *Digital Investigation* **28**, 56–69.

[9] Markanovic, M. and Persson, S. [2014], 'Trusted memory acquisition using uefi'.
URL: http://www.diva-portal.org/smash/get/diva2:830892/FULLTEXT01.pdf

[10] Martignoni, L., Fattori, A., Paleari, R. and Cavallaro, L. [2010], Live and trustworthy forensic analysis of commodity production systems, *in* S. Jha, R. Sommer and C. Kreibich, eds, 'Recent Advances in Intrusion Detection, 13th International Symposium, RAID 2010, Ottawa, Ontario, Canada, September 15-17, 2010. Proceedings', Vol. 6307 of *Lecture Notes in Computer Science*, Springer, pp. 297–316.
URL: *https://doi.org/10.1007/978-3-642-15512-3_16*

[11] Moser, A. and Cohen, M. I. [2013], 'Hunting in the enterprise: Forensic triage and incident response', *Digit. Investig.* **10**(2), 89–98.
URL: *https://doi.org/10.1016/j.diin.2013.03.003*

[12] Oleksiuk, D. [2015], 'Building reliable SMM backdoor for UEFI based platform'.
URL: *http://blog.cr4.sh/2015/07/building-reliable-smm-backdoor-for-uefi.html*

[13] Palutke, R. and Freiling, F. [2018], 'Styx: Countering robust memory acquisition', *Digital Investigation* **24**.

[14] Rowlingson, R. [2004], 'A ten step process for forensic readiness', *International Journal of Digital Evidence* **2**(3), 1–28.

[15] Sparks, S. and Butler, J. [2005], 'Shadow walker: Raising the bar for rootkit detection', *Black Hat Japan* **11**(63), 504–533.

[16] Stüttgen, J. and Cohen, M. [2013], 'Anti-forensic resilient memory acquisition', *Digital Investigation* **10**, S105–S115.

[17] Sylve, J. [2012], 'LiME'. https://github.com/504ensicsLabs/LiME.

[18] TianoCore [2020], 'EDK II Project'. https://github.com/tianocore/edk2.

[19] TianoCore [n.d.], 'EDK II Driver Writer's Guide'. https://edk2-docs.gitbook.io/edk-ii-uefi-driver-writer-s-guide/5_uefi_services.

[20] Vömel, S. and Freiling, F. C. [2011], 'A survey of main memory acquisition and analysis techniques for the windows operating system', *Digit. Investig.* **8**(1), 3–22.
URL: *https://doi.org/10.1016/j.diin.2011.06.002*

[21] Vömel, S. and Freiling, F. C. [2012], 'Correctness, atomicity, and integrity: defining criteria for forensically-sound memory acquisition', *Digital Investigation* **9**(2), 125–137.